**PREM KRISHNA CHETTRI**

**Computer Architecture Assignment 1 Submission Date: 05st Oct ‘15**

**Solution 1** :

From the given problem, we can calculate the the CPI of each part of the instructions as follows :-

1. For the fraction i of the instruction fetches result in an I-cache miss and each miss requires B cycles to deliver the required instruction.

CPI = i/N \*B.

1. For the fraction m of all instructions are LOADs and a fraction q of these LOAD instructions experience a D- cache miss. A D-cache miss requires M cycles to fetch the data from the RAM into the D-cache

CPI = q / ( m / N) \* M = q / m \* M \* N.

1. For a fraction n of all instructions are STOREs and a fraction p of these STORE instructions experience a D-cache miss. A D-cache miss requires M cycles to fetch the data from the RAM into the D-cache.

CPI =p / (n / N) \* M = p / n \* M \* N.

1. For another fraction b of the instructions are branches and a fraction t of the branches are taken (that is, result in a non-consecutive instruction access). Each such taken branch introduces a S-cycle bubble.

CPI = t / (b / N) \* S = t / b \* S \* N .

1. Finally a fraction d of the instructions are stalled for an extra R cycles due to data-dependencies.

CPI = d / N \* R

So now the effective CPI realized for the processing of this instruction is the summation of total CPI by 100. So it is equivalent to

Effective CPI = (I / N) \* B + (q / m) \* M \* N + (t / b ) \* N \* S + (d / N) \* R + (d / N) \* R.

**Solution 2** :

The changes I made to an existing design to incorporate MUL operations required demultiplexure , multiplexure and an addition of one more input port to the register file for decoding 2 instructions to decode two registers during multiplication.

Firstly, on fetch stage. For ADD of the type ADD Rx,Ry,#C, we can send address of register Ry to be decoded to register file and in case of type MUL Rx,Ry,Rz , we can send an addresses of register Ry, Rz to be decoded to the register file. Once, I’ll have values of registers available, I can use them to feed on to ALU. One of an operand will always be a value of register Ry, whereas the other value to ALU have to be determined based on the third source of an instruction. So we need to use an OR gate to determine if it is either of #c or the decoded value of Rz (whichever is available) for which we will have to take one more cycle in D / RF stage.

Meanwhile, the decoder had decoded the type of the operation and now I can ask ALU to perform the operation specified by instruction and provide output.

On execute stage, we will perform an ALU operation based on the inputs available and the decoded information. Also, we will perform the address calculation of destination register Rz to be used during Write back cycle.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| INSTRUCTION \ Stage | F stage | D/RF stage | EX stage | MEM stage | WB stage |
| Reg–to–reg: ADD rdest, rsrc1, rsrc2 | Read instruction from memory | Decode instruction; Read rsrc1 and rsrc2 from the RF | Add contents of rsrc1 and rsrc2; Set condition code flags | Hold on to the result produced in the EX stage in the previous cycle; | Write result to register rdest |
| Load indexed–literal offset: LDI rdest, rsrc1, <literal> | Read instruction from memory | Decode instruction; Read rsrc1; sign extend <literal> | Add contents of rsrc1 and sign extended literal; Set condition code flags; | Read the contents of the memory location whose address was computed in the previous cycle in the EX stage | Write the data read out from memory to the register rdest |
| Load indexed–reg. offset: LDIR rdest, rsrc1, rsrc2 | Read instruction from memory | Decode instruction; Read rsrc1 and rsrc2 from the RF | Add contents of rsrc1 and rsrc2; Set condition code flags | Read the contents of the memory location whose address was computed in the previous cycle in the EX stage | Write the data read out from memory to the register rdest |
| Store indexed–literal offset: STI rsrc1, rsrc2, <literal> | Read instruction from memory | Decode instruction; Read rsrc1 and rsrc2; sign extend <literal> | Add contents of rsrc2 and sign extended literal; Set condition code flags | Write the contents of rsrc1 (read out in the D/RF stage) to the memory location whose address was com- puted in the EX stage in the previous cycle | No action |

**Solution 3** :

As in problem 2 , we will use the OR gate to separate the values amount different operands in D /RF stage. We will use a shifter in Execute stage and call as EX2 stage.

In D/ RF stage, we will decode the value of src1 and src2 of the shift instruction of type ADDSH <dest> <src1><src2> <# shift> by the help of 2 input port register file. Alongside, we use the decoder to decode the instruction type from the instruction register. We will use the OR gate here again to segregate the <sr2> decoded value with the values of the fixed literal like ADD r5, r1 m #c. This introduces an additional cycle delay in D / RF stage but is needed to make sure that we will pick correct operand between ADDSH <src2> and ADD <#c> while performing ALU operation.

In Execute stage, we remain with the original design of the given APEX datapath and name it as EX1 stage. However, once we have an output <sum in our case>, we will use that value and pass on to EX2 stage which basically does the shift operation.

In EX2 stage of execution, we will take the decoded values of ADDSH instruction from latch and will feed to the shifter with the following inputs:- The “direction of shift” information and the “logical / ALU shift” operation information from the latch. Also, we use the <#shift> value from the shift instruction that we have decoded in D / RF stage and perform the shift operation for the output sum from the EX1 stage of pipeline (only if we have ADDSH instruction decoded by our decoder). The result of which is going to be the final value of the total operation and we will use the same datapath as the original design for the rest of the writeback opertions.

1. ADDSH operation : - As discussed above, for ADDSH , we will decode the value of SR1, SR2 and feed those values to the ALU. On, decoder we decode the ADDSH which will provide the information of direction of shift and and logical or ALU operation to be performed while shifting and now. In EX1 stage of execute , we will execute the addition operation with the help of ALU and we pass that value to EX2 which will perform the shift operation. Here, we will receive , the added sum value from the EX1 stage, #shift value from D / RF stage and the “direction” , “type of shift” operation from the decoded latch and we perform shift on this EX2 stage. Once, this is done we will pass the value to Memory stage which finally gets passed to register for Write Back operation.
2. ADD operation : Add operation will we same as from our original datapath. We will decode the instruction and register value and pass them to EX1 of execute stage. In here, we will perform the addition and we pass the value to EX2 stage, but as our decode instruction will not have any #shift value associated as well as the latches wont have any information for direction / type of shift, the calculated sum will simply pass as it is to the next stages to be finally written back to register file.
3. LOAD operation : LDI / LDIR both of these two load instruction will perform as in its original design as we won’t do any shift operation in EX2 stage of execution. We simply pass on the value to the next stage to be written back to register file.